PRISMS-PF User Guide (v2.0)

Stephen DeWitt (prismsphasefield.dev@umich.edu)

August 10, 2017

Contents

1 Overview

PRISMS-PF is an open-source finite element simulation code with a focus on solving equations from phase field models. It is being developed as part of the PRISMS Center at the University of Michigan. PRISMS-PF is built on top of the popular and powerful deal.II finite element library. PRISMS-PF was developed to make high performance phase field simulation capabilities available to the wider scientific community with an easy-to-use and easy-to-learn interface. Benchmark tests show that even without its adaptive meshing capabilities PRISMS-PF is competitive with specialized finite difference codes written in Fortran with MPI parallelization in terms performance, and in some cases runs several times faster. PRISMS-PF can be used to solve a wide variety of systems of coupled parabolic partial differential equations (e.g. the diffusion equation) and elliptic partial differential equations (e.g. Poisson's equation). With PRISMS-PF, you have access to adaptive meshing and parallelization that has been demonstrated on over a thousand processors. Moreover, the matrix-free framework from deal.II allows much larger than typical finite element programs – PRISMS-PF has been used for calculations with over one *billion* degrees of freedom.

This user guide starts with instructions for downloading PRISMS-PF and install the necessary prerequisites. Next are instructions for running the built-in example applications and visualizing the results. A detailed look at the input files follows, and the guide finishes with some instructions on how to create your own PRISMS-PF applications.

If you run into any issues not covered by this guide, or have any questions, please send an email to the PRISMS-PF email list: prisms-pf-users@googlegroups.com. If you'd like to contact the PRISMS-PF developers directly, send an email to: prismsphaseField.dev@umich.edu.

2 Downloading deal.II and PRISMS-PF

Before downloading PRISMS-PF itself, one should install CMake and deal.II. CMake can be downloaded at: https://cmake.org/download, which also has installation instructions. Be sure to add CMake to your path by entering:

```
$ $ PATH="/path/to/cmake/Contents/bin":"$PATH"
```

filling in the path to the installation of CMake (for example, on Mac OS, the default installation path is /Applications/CMake.app/Contents/bin). For convience, we recommend adding this line to your bash profile. Your bash profile can be opened via the following terminal command

```
$ vi ~/.profile
```

Deal.II can be downloaded at https://www.dealii.org/download.html, which also has instructions. The deal.II installation process depends on your operating system. PRISMS-PF has been tested with Deal.II versions 8.3.0 through 8.5.0. We recommend using deal.II 8.4.2, if possible. If you use deal.II 8.5.0, expect to recieve a number of warnings during compilation (discussed in Sec. 3) about functions used in PRISMS-PF that are deprecated in deal.II 8.5.0.

2.1 Installing deal.II for Mac OS X

A binary package for version 8.4.2 (and other versions) is available at https://github.com/dealii/dealii/releases. The binary package includes all the libraries that you need (excluding CMake). To install deal.II open the .dmg file and drag "deal.II.app" into your Applications folder. Then, open "deal.II.app", which will open a Terminal window and set the necessary environment variables. You may need to modify your security settings so that your operating system will let you open the application (in System Preferences, select Security & Privacy, then under the general tab choose to allow apps downloaded from anywhere; don't click "Open deal.II.app anyway", we have found that doing so doesn't actually launch deal.II.app).

In some cases, deal.II.app will not open, possibly freezing at the "Verifying..." stage. If this happens, right click on it and select "Show Package Contents". Then go to "/Contents/Resources/bin/" and launch "dealii-terminal". This should launch the Terminal window that sets the environment variables.

We have found that for some older versions of Mac OS are incompatible with deal.II version 8.4.2 (due to the version of the Clang compiler that is installed by default). In those cases we recommend downloading this version 8.3.0 package:
https://github.com/dealii/dealii/releases/download/v8.3.0/dealii-8.3.0.nocxx14.dmg.

2.2 Installing deal.II for Linux

Unfortunately, on a Linux machine you will likely have to compile and install deal.II from the source. The source code for version 8.4.2 can be downloaded at https://github.com/dealii/dealii/releases. Detailed installation directions can be found at https://www.dealii.org/8.5.0/readme.html. For some versions of Linux, packages are available (Gentoo, Debian testing (stretch), and Ubuntu 16.10+). See https://www.dealii.org/download.html for details.

Below we list instructions for installing deal.II and its prerequisites both manually and using candi, an automatic deal.II installation script. For now, we recomend the traditional route, because we haven't been able to get candi to consistently work. Recently, deal.II has been made available on Linuxbrew. We have not tried this yet, but it may be a good option. Details can be found at https://github.com/dealii/dealii/wiki/deal.II-on-Homebrew—Linuxbrew.

2.2.1 Traditional Installation (Recommended assuming no package is available)

Before building deal.II, you will need to install two libraries, MPI and p4est. Make sure you have both libraries installed before installing deal.II, or else you will have to reinstall deal.II! Many computing clusters already have these libraries installed. MPI libraries (such as MPICH2) are often available through package managers (e.g. yum, apt, pkg_add, port, brew), installing it using your package manager of choice is likely the simplest option. If not, you can find binaries and the source code at http://www.mpich.org/downloads/, as well as instructions for installation. The p4est library may also be available through your package manager. If not, you can download the latest release tarball at http://www.p4est.org/. To install, unzip the archive, move to the p4est root directory and enter the following commands at the command line:

```
$ ./configure ––enable–mpi; make; make install
```

Users have found that this approach works better than the setup script "p4est-setup.sh" that is on the deal.ii website. More detailed installation instructions can be found in the "INSTALL" and "README" files in the p4est root directory.

With MPI and p4est installed, you are ready to install deal.II. Open the archive you downloaded from the deal.II website. From the root deal.II directory enter the following commands:

```
$ mkdir build
$ cd build
$ cmake −DP4EST_DIR=/path/to/installation/dir −DDEAL_II_WITH_P4EST=ON
−DDEAL_II_WITH_MPI=ON −DCMAKE_INSTALL_PREFIX=/path/to/install/dir
$ make install
```

where "/path/to/install/dir" is the path to where you want to install deal.II. After installation is complete (which can take take up to an hour), open the file "summary.log". The second section of the log file lists the configured features. Ensure that DEAL_II_WITH_MPI and DEAL_II_WITH_P4EST say either "set up with bundled packages" or "set up with external dependences". If either is listed as "OFF", then deal.II was unable to find MPI or p4est. If this is the case, double check that MPI and p4est were installed correctly.

2.2.2   Using candi

An alternative route is to use the following candi (compile and install) script to install deal.II and its prerequisites:
https://github.com/koecher/candi

If it works, it should be much simpler than the traditional route described above. However in limited attempts, we have not been able to successfully install deal.II with candi. After more testing and/or consultation with the developer, we hope to make this a plausible option.

2.3   Downloading PRISMS-PF

PRISMS-PF is available for download at our GitHub site: https://github.com/prisms-center/phaseField. The recommended method for downloading the PRISMS-PF source code is through git. Using a Linux/Unix terminal, go to the directory where you want PRISMS-PF to be located. To clone the repository, type:

```
$ git clone https://github.com/prisms−center/phaseField.git
```

Git will then download the PRISMS-PF source code into a directory entitled "phaseField". A resource for learning to use Git can be found at:
https://git-scm.com/book.

If you prefer not to use Git, a zip file containing the PRISMS-PF source code can be downloaded at:
https://github.com/prisms-center/phaseField/releases.

3 Running the Example Applications

3.1 The Example Applications

After deal.II and PRISMS-PF are downloaded, you can run the pre-built PRISMS-PF example applications. At this time, the example applications include:

- allenCahn: An implementation of the Allen-Cahn equation for two phases. (2D)

- allenCahn_pfield: Like allenCahn, but loads in initial conditions using PRISMS IntegrationTools. (2D)

- cahnHilliard: An implementation of the Cahn-Hilliard equation for two phases. (2D)

- cahnHilliardWithAdaptivity: Like cahnHilliard, but with adaptive meshing enabled. (2D)

- coupledCahnHilliardAllenCahn: An implementation of the coupled Cahn-Hilliard/Allen-Cahn set of equations. (2D)

- CHAC_anisotropy: Coupled Cahn-Hilliard/Allen-Cahn equations with weakly anisotropic interfacial energy. (2D)

- CHAC_anisotropyRegularized: Like CHAC_anisotropy, but with a regularization term to permit strongly anisotropic interfacial energy. (2D)

- fickianDiffusion: An implementation of the diffusion equation with a time-dependent source term. (2D)

- mechanics: An implementation of linear elasticity for a material in uniaxial tension. (3D)

- CHiMaD_benchmark1a: An implementation of the CHiMaD spinodal decomposition benchmark problem. (2D)

- CHiMaD_benchmark2a: An implementation of the CHiMaD Ostwald ripening benchmark problem. (2D)

- CHiMaD_benchmark6a: An implementation of the CHiMaD electrochemistry benchmark problem. (2D)

- CHiMaD_benchmark6b: An implementation of the CHiMaD electrochemistry benchmark problem with a curved domain. (2D)

- dendriticSolidification: An implementation of a solidification model for a pure material resulting in the growth of a dendrite. (2D)

- eshelbyInclusion: An implementation of linear elasticity for a spherical inclusion. (3D)

- grainGrowth: An implementation of ten coupled Allen-Cahn equations simulating grain growth in two dimensions. (2D)

- precipiateEvolution: An implementation of the coupled Cahn-Hilliard/Allen-Cahn/Linear Elasticity equations often used in phase field simulation of precipitate evolution. (2D)

- precipiateEvolution_pfunction: Like precipiateEvolution, but loads inputs using PRISMS IntegrationTools. (2D)

- singlePrecipitateKKS: Similar to precipiateEvolution, but uses the KKS model rather than the WBM model for the free energy functional. (3D)

- nucleationModel: KKS precipitation model that makes use of the PRISMS-PF explicit nucleation capabilities. (2D)

- preferential_nucleationModel: Like nucleationModel, but with a zone with an increased nucleation rate to simulate a grain boundary. (2D)

A directory for each of these applications can be found in the applications directory (i.e. phaseField/applications). The applications contain a formulation file giving the governing equations. In addition to the 21 applications listed above, some application names may be preceded by an underscore. The underscore is used to denote applications that are still under active development.

3.2 Compiling the Core PRISMS-PF Library

After downloading PRISMS-PF, we recommend that you compile the core library (see note at the end of this section). To do so, enter the root PRISMS-PF directory from the directory where you downloaded it to, run CMake, and then compile the library:

```
$ cd phaseField
$ cmake .
$ make −j 8
```

In the last command, the number following "-j" is the number of threads used during compilation. The general rule of thumb is to pick a number $1.5\times$ the number of processors available. This process should take a minute or two and will compile both the "debug" and "release" versions of the core PRISMS-PF library (see 3.7).

Note: This step can be skipped and the core library will be compiled the first time that you compile one of the applications. However, the compilation will be on a single thread and thus will take much longer than if you follow the instructions above and use multiple threads.

3.3 Running the Allen-Cahn Example Application

From the "phaseField" directory one can run the Allen-Cahn example application through to following terminal commands:

```
$ cd applications/allenCahn/
$ cmake .
$ make debug
$ mpirun −n 1 main
```

The first command moves from the "phaseField" directory to the directory of the Allen-Cahn example. The second command checks that core PRISMS-PF library has been compiled, (re-)compiles it if necessary, and creates a *makefile* using CMake. The third command compiles the executable in "debug" mode, which enables a number of exception checks in the code and adds debugging information that can be used by a debugger (e.g. gdb). The fourth command runs the program using a single processor.

As the program runs, information from each time step outputs to the terminal window. After the simulation is complete, a summary the time taken in a few major sections of the code and the total wall time is printed to the terminal window.

Here is a screenshot of typical output from CMake as you create the *makefile*:

```
vega:allenCahn stephendewitt$ cmake .
-- Using the deal.II-8.4.1 installation found at /Applications/deal.II.app/Contents/Resources
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_add_test.cmake
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_initialize_cached_variables.cmake
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_invoke_autopilot.cmake
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_pickup_tests.cmake
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_query_git_information.cmake
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_setup_target.cmake
-- The C compiler identification is AppleClang 7.3.0.7030031
-- The CXX compiler identification is AppleClang 7.3.0.7030031
-- Check for working C compiler: /Applications/deal.II.app/Contents/Resources/opt/openmpi-1.10.2/bin/mpicc
-- Check for working C compiler: /Applications/deal.II.app/Contents/Resources/opt/openmpi-1.10.2/bin/mpicc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /Applications/deal.II.app/Contents/Resources/opt/openmpi-1.10.2/bin/mpic++
-- Check for working CXX compiler: /Applications/deal.II.app/Contents/Resources/opt/openmpi-1.10.2/bin/mpic++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Using the deal.II-8.4.1 installation found at /Applications/deal.II.app/Contents/Resources
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_add_test.cmake
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_initialize_cached_variables.cmake
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_invoke_autopilot.cmake
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_pickup_tests.cmake
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_query_git_information.cmake
-- Include macro /Applications/deal.II.app/Contents/Resources/share/deal.II/macros/macro_deal_ii_setup_target.cmake
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/stephendewitt/Documents/workspace/PRISMS_workspace/phaseField_dev
[ 50%] Built target prisms_pf
[100%] Built target prisms_pf_debug
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/stephendewitt/Documents/workspace/PRISMS_workspace/phaseField_dev/applications/allenCahn
vega:allenCahn stephendewitt$ 
```

Don't worry if the output isn't exactly the same as what you see, the details of some of the messages depend on your operating system and which compilers you have installed. The important part is that the bottom three messages are "Configuring done", "Generating done", and "Build files have been written to: ...". In the future, entering "$ cmake ." will result in a shorter set of messages because CMake caches some variables from the last time it was run. As a result, you can omit the CMake step for future simulations as long as the path name to your current directory is unchanged and your installation of deal.II is unchanged.

Here is a screenshot of typical output from the compiler as you compile the executable:

Depending on your version of deal.II, different warnings may appear as you compile. Common warnings include the use of functions that deal.II has marked as depricated (as in the screenshot above) and unused type definitions. In this case, PRISMS-PF uses these functions for backward compatability with deal.II version 8.4.x. We will switch to the updated functions in the near future.

Once the simultation is complete, the terminal output at the end of the simulation should look like:

```
time increment:19998  time: 19.998
wall time: 0.00685596s

time increment:19999  time: 19.999
wall time: 0.0065410ls

time increment:20000  time: 20
wall time: 0.00612307s
Output written to:solution-020000.pvtu

Energy: 182.087
Energy Components: 74.7191 107.367 0

currentTime>=timeFinal. Ending time stepping


+-------------------------------------------+-----------+-----------+
| Total wallclock time elapsed since start  |    162s   |           |
|                                           |           |           |
| Section                                   | no. calls | wall time | % of total |
+-------------------------------------------+-----------+-----------+-----------+
| matrixFreePDE: computeEnergy    |       21 |   3.64s |    2.3% |
| matrixFreePDE: computeRHS       |    20000 |   123s  |    76%  |
| matrixFreePDE: initialization   |        1 |   1.81s |    1.1% |
| matrixFreePDE: output           |       21 |   13s   |    8%   |
| matrixFreePDE: solve            |        1 |   160s  |    99%  |
| matrixFreePDE: solveIncrements  |    20000 |   140s  |    86%  |
+-------------------------------------------+-----------+-----------+-----------+

vega:allenCahn stephendewitt$ []
```

3.4   What Can Go Wrong

If you were able to enter all of the commands in the previous section and get output similar
to the screenshots, congratulations! you just ran your first PRISMS-PF simulation. If not, you
may be experiencing one of the common issues listed below.

If CMake gives an error message like this:

```
vega:allenCahn stephendewitt$ cmake
Usage

  cmake [options] <path-to-source>
  cmake [options] <path-to-existing-build>

Specify a source directory to (re-)generate a build system for it in the
current working directory.  Specify an existing build directory to
re-generate its build system.

Run 'cmake --help' for more information.

vega:allenCahn stephendewitt$ []
```

Then you likely forgot the period at the end of the command "$ cmake .".

If CMake gives an error message like this:

```
[stvdwtt@flux-login2 allenCahn]$ cmake .
CMake Error at CMakeLists.txt:23 (MESSAGE):

  *** Could not locate deal.II.  ***


  You may want to either pass a flag -DDEAL_II_DIR=/path/to/deal.II to cmake

  or set an environment variable "DEAL_II_DIR" that contains this path.


-- Configuring incomplete, errors occurred!
[stvdwtt@flux-login2 allenCahn]$ []
```

CMake cannot find your installation of deal.ii. This issue is probably caused by the lack of an environment variable pointing to the directory containing your deal.II library. You can check this with the following command:

$ echo $DEAL_II_DIR

The terminal should then output the path to the deal.II library. For example in Mac OS, the deal.II directory may be "/Applications/deal.II.app/Resources". If DEAL_II_DIR contains a path, go there to see if deal.II is actually installed there. If DEAL_II_DIR is incorrect or empty, you should set it to the directory of the correct path of your deal.II installation with the following command:

$ export DEAL_II_DIR=/path/to/dealii

Environment variables are erased when you close your shell. To have this path set every time you open a shell (i.e. every time you open a new terminal window), you can add the command above to your shell profile (e.g. .bashrc if you use a bash shell). If you are still having problems, there may be an issue with your deal.II installation. Please consult the deal.II website for instructions.

If CMake cannot successfully detect a C++ compiler, it will generate an error message. The most common cause for this is that the machine runs the Mac OS operating system with an outdated version of the Clang compiler. Upgrading your OS to version 10.11 or newer, updating Xcode, and (re)installing the Xcode command line tools may help. Alternatively, you can install a certain version of the deal.II package that was developed to sidestep this issue:
https://github.com/dealii/dealii/releases/download/v8.3.0/dealii-8.3.0.nocxx14.dmg

Most of issues users have had are during the CMake step. If the fixes suggested above don't work for your or you have an issue not covered by this list, please contact the PRISMS-PF users list: prisms-pf-users@googlegroups.com. If you are not already on the list, please submit a join request through Google Groups or send an email with "SUBSCRIBE" in the subject line

to prismsphasefield.dev@umich.edu. As users come across new issues, we will add them (and suggested fixes) to this section.

3.5 Visualizing the Results of the Simulation

Once you have successfully run a simulation, you will likely want to visualize the results. PRISMS-PF output files are generated in the popular VTK format, as a series of *.vtu and *.pvtu files. Two common open-soure, multi-platform visualization tools for these types of files are VisIt and ParaView. Instructions for downloading this software can be found at their respective websites:
VisIt: https://wci.llnl.gov/simulation/computer-codes/visit/
ParaView: http://www.paraview.org/

To get you started, here is a brief tutorial on how to use VisIt to visualize your simulation results. For more detailed instructions, please consult the VisIt manual:
https://wci.llnl.gov/simulation/computer-codes/visit/manuals

After launching the VisIt application, click "Open" and find the directory for the Allen Cahn example:
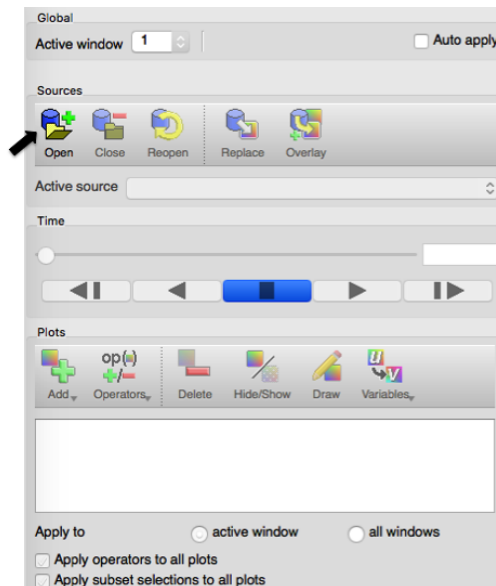
and select "solution-*.pvtu:
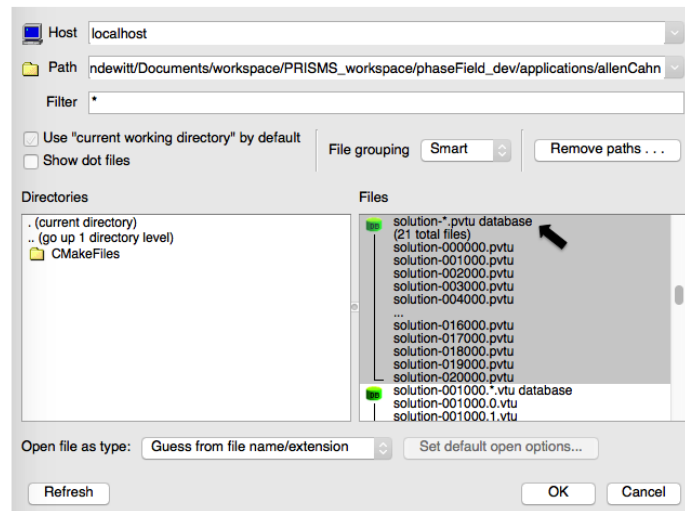
Next, click "Add", hover over "Pseudocolor", and select "n":
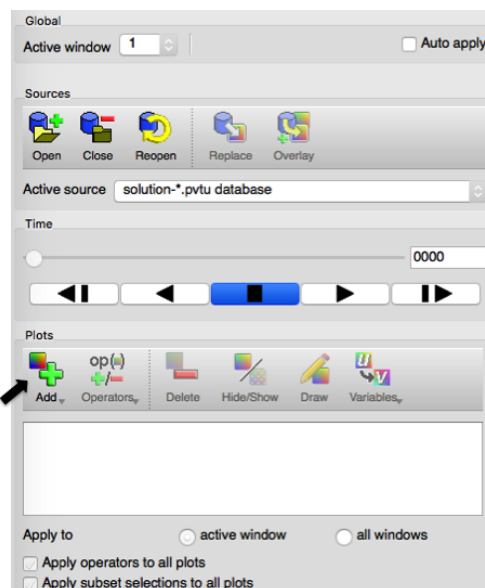
Next, click "Draw" to make a plot:
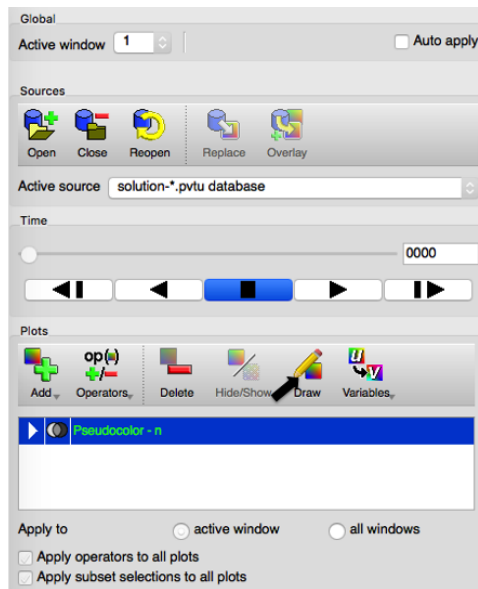
The VisIt window will now have a plot of the initial condition of the order parameter:
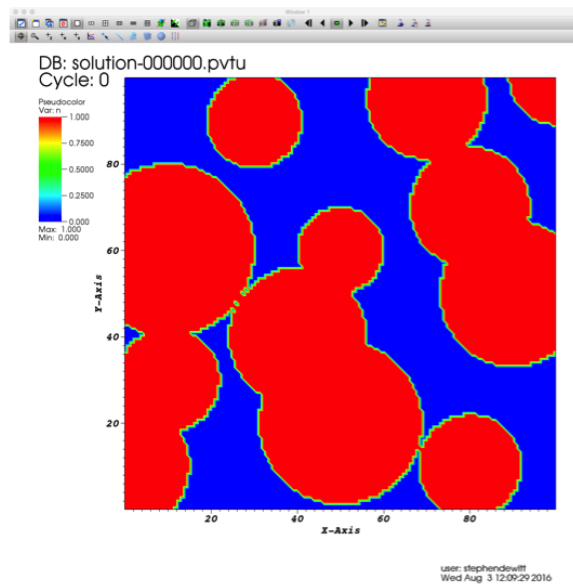
The result at other time steps can be visualized by dragging the time bar, or using the controls directly below the time bar. Dragging the time bar to the end will display the final result of the simulation:
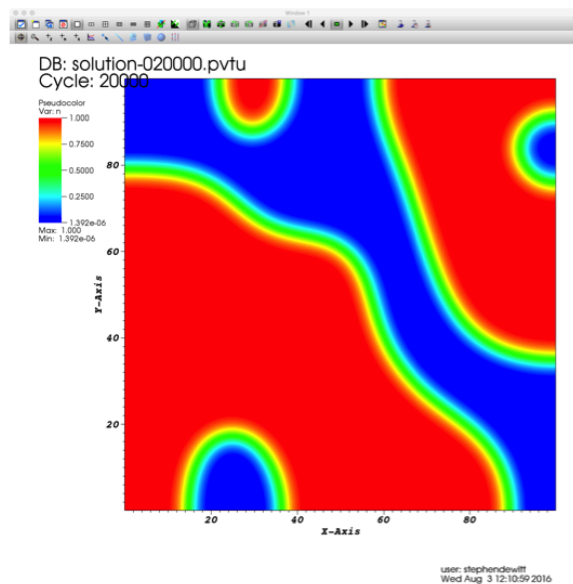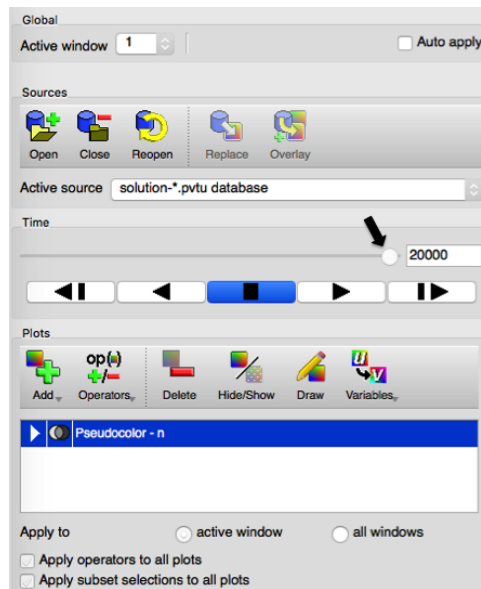
VisIt has a wide variety of capabilities for visualizing 1D, 2D, and 3D data, including postprocessing of output fields (e.g. to obtain derivatives of output fields or the result of mathematical expressions involving one or more output field). VisIt also has a powerful Python interface to provide scripting capabilities. More more instructions on how to use these, and other, features of VisIt, please consult the VisIt manual.

3.6 Running the Other Example Applications

Running the other example applications is as simple as going to the directory for the application of interest and repeating the steps in Section 3.3. For example, to run the Cahn-Hilliard example application, when you are currently in the directory for the Allen-Cahn example application, you would type the following commands:

```
$ cd ../cahnHilliard/
$ cmake .
$ make debug
$ mpirun -n 1 main
```

3.7 Running in Release Mode and with Multple Processors

Once you are sure that your code works as expected, you can compile it in "Release Mode", which is approximately 10x faster than "Debug Mode". However, no exceptions are checked in Release Mode and therefore the code may compile and run even if it contains several errors. Therefore, we strongly recommend that all new code is tested in Debug Mode before switching to Release Mode. To compile in Release mode, replace "$ make debug" with "$ make release".

One of the strengths of PRISMS-PF is that it can be run in parallel with almost no extra effort from the user. To run a simulation in parallel, replace the "1" in the "mpirun" command with the desired number of processor cores. The deal.II packages on their website already contain the MPI library, so no extra software has to be downloaded to use multiple cores.

From the directory of an example application, a simulation can be run in Release Mode on 4 cores using the following commands:

```
$ cmake .
$ make release
$ mpirun -n 4 main
```

4 Structure of a PRISMS-PF Application

4.1 Files in the Application Directories

Each of the example application directories should contain at least eight files:

- parameters.in
- equations.h
- ICs_and_BCs.h
- postprocess.h
- customPDE.h
- main.cc
- formulation.pdf

17

- CMakeLists.txt

The following section (Sec. 5) will discuss the structure and options for the input file, parameters.h. Users who do not need to change the governing equations or make other fundamental changes to the application will only need to modify this file.

Section 6 discusses the structure of the files that define an application: equations.h, ICs_and_BCs.h, postprocess.h, customPDE.h, and main.cc. Users who want to substantially modify existing PRISMS-PF applications or create their own will find this section useful.

The final two files in each application directory are formulation.pdf and CMakeLists.txt. The file formulation.pdf describes the mathematical formulation of the problem solved by the application. Finally, the file CMakeLists.txt is used by CMake to generate the makefile. Most users will not have any reason to change CMakeLists.txt.

5 The Input File: parameters.in

Most PRISMS-PF users will spend the majority of their time interacting with the input file, parameters.h. This file allows users to specify the computational domain, the mesh, the time step parameters, the boundary conditions, the output, and model constants (and more). This file is read as a text file, so modifications to it do not require recompilation of the application.

Here is an example of an input file from the allenCahn application:

```
# Parameter list for the Allen−Cahn example application
# Refer to the PRISMS−PF manual for use of these parameters in the source code.


# =================================================================================
# Set the number of dimensions (2 or 3 for a 2D or 3D calculation)
# =================================================================================
set Number of dimensions = 2


# =================================================================================
# Set the length of the domain in all three dimensions
# (Domain size Z ignored in 2D)
# =================================================================================
# Each axes spans from zero to the specified length
set Domain size X = 100
set Domain size Y = 100
set Domain size Z = 100


# =================================================================================
# Set the element parameters
# =================================================================================
# The number of elements in each direction is 2^(refineFactor) * subdivisions
# Subdivisions Z ignored in 2D
# For optimal performance, use refineFactor primarily to determine the element size
set Subdivisions X = 1
set Subdivisions Y = 1
set Subdivisions Z = 1

set Refine factor = 8

# Set the polynomial degree of the element (allowed values: 1, 2, or 3)
set Element degree = 1


# =================================================================================
# Set the time step parameters
# =================================================================================
# The size of the time step
set Time step = 1.0e−2

# The simulation ends when either the number of time steps is reached or the
```

```
# simulation time is reached.
set Number of time steps = 5000

# ================================================================
# Set the boundary conditions
# ================================================================
# Set the boundary condition for each variable, where each variable is given by
# its name, as defined in equations.h. The four boundary condition
# types are NATURAL, DIRICHLET, NON_UNIFORM_DIRICHLET and PERIODIC. If all
# of the boundaries have the same boundary condition, only one boundary condition
# type needs to be given. If multiple boundary condition types are needed, give a
# comma-separated list of the types. The order is the miniumum of x, maximum of x,
# minimum of y, maximum of y, minimum of z, maximum of z (i.e left, right, bottom,
# top in 2D and left, right, bottom, top, front, back in 3D). The value of a
# Dirichlet BC is specfied in the following way -- DIRCHILET: val -- where 'val' is
# the desired value. If the boundary condition is NON_UNIFORM_DIRICHLET, the
# boundary condition should be specified in the appropriate function in 'ICs_and_BCs.h'.
# Example 1: All periodic BCs for variable 'c'
# set Boundary condition for variable c = PERIODIC
# Example 2: Zero-derivative BCs on the left and right, Dirichlet BCs with value
# 1.5 on the top and bottom for variable 'n' in 2D
# set Boundary condition for variable n = NATURAL, NATURAL, DIRICHLET: 1.5, DIRICHLET: 1.5

set Boundary condition for variable n = NATURAL

# ================================================================
# Set the model constants
# ================================================================
# Set the user-defined model constants, which must have a counter-part given in
# customPDE.h. These are most often used in the residual equations in equations.h,
# but may also be used for initial conditions and nucleation calculations. The type
# options currently are DOUBLE, INT, BOOL, TENSOR, and [symmetry] ELASTIC CONSTANTS
# where [symmetry] is ISOTROPIC, TRANSVERSE, ORTHOTROPIC, or ANISOTROPIC.

# The mobility, MnV in equations.h
set Model constant MnV = 1.0, DOUBLE

# The gradient energy coefficient, KnV in equations.h
set Model constant KnV = 2.0, DOUBLE

# ================================================================
# Set the output parameters
# ================================================================
# Type of spacing between outputs ("EQUAL_SPACING", "LOG_SPACING", "N_PER_DECADE",
# or "LIST")
set Output condition = EQUAL_SPACING

# Number of times the program outputs the fields (total number for "EQUAL_SPACING"
# and "LOG_SPACING", number per decade for "N_PER_DECADE", ignored for "LIST")
set Number of outputs = 5

# The number of time steps between updates being printed to the screen
set Skip print steps = 1000
```

The syntax for setting each input parameter is:

```
set [parameter name] = [parameter value]
```

The pound symbol (#) is used for comments, and any text after it is ignored by the file parser.

The following table lists all of the input parameters, seperated into the same groupings in the input file above:

Dimensionality				
Name	*Options*	*Required*	*Default*	*Description*
Number of dimensions	2, 3	yes	n/a	The number of dimensions for the simulation.

Compuational Domain				
Name	*Options*	*Required*	*Default*	*Description*
Domain size X	Any positive real number	yes	n/a	The size of the domain in the x direction.
Domain size Y	Any positive real number	yes	n/a	The size of the domain in the y direction.
Domain size Z	Any positive real number	yes	n/a	The size of the domain in the z direction.

Element Parameters				
Name	*Options*	*Required*	*Default*	*Description*
Subdivisions X	Any positive integer	no	1	The number of mesh subdivisions in the x direction to control the element aspect ratio. The mesh size is $2^{(RefineFactor)} \times$ Subdivisions in each direction.
Subdivisions Y	Any positive integer	no	1	The number of mesh subdivisions in the y direction to control the element aspect ratio.The mesh size is $2^{(RefineFactor)} \times$ Subdivisions in each direction.
Subdivisions Z	Any positive integer	no	1	The number of mesh subdivisions in the z direction to control the element aspect ratio. The mesh size is $2^{(RefineFactor)} \times$ Subdivisions in each direction.
Refine factor	Any non-negative integer	yes	n/a	The number of initial refinements of the mesh. The mesh size is $2^{(RefineFactor)} \times$ Subdivisions in each direction. While in principle the mesh could be entirely controlled by the number of subdivisons, computational performance is best when the majority of the refinement is done via the Refine factor.
Element degree	1, 2, 3	no	1	The polynomial order of the elements. The spatial order of accuracy is one plus the degree.

Mesh Adaptivity				
Name	*Options*	*Required*	*Default*	*Description*
Mesh adaptivity	Boolean	no	false	Controls whether mesh adaptivity is enabled.
Max refinement level	Any non-negative integer	no	-1	The maximum number of local refinements during adaptive meshing. This parameter does not need to be specified if mesh adaptivity is disabled, but the default value will cause an error if mesh adaptivity is enabled.
Min refinement level	Any non-negative integer	no	-1	The minimum number of local refinements during adaptive meshing. This parameter does not need to be specified if mesh adaptivity is disabled, but the default value will cause an error if mesh adaptivity is enabled.
Refinement criteria fields	Comma separated list of variable names	no	0	The indices of the variables that will determine the mesh refinement. The variable names are determined by the names given in equations.h.
Refinement window max	Comma separated list of real numbers	no		The mesh refines where the specified variables are between an upper and lower bound. This specifies the upper bound.
Refinement window min	Comma separated list of real numbers	no		The mesh refines where the specified variables are between an upper and lower bound. This specifies the lower bound.
Steps between remeshing operations	Positive integer	no	1	The number of time steps between mesh refinement operations.

Time Stepping				
Name	*Options*	*Required*	*Default*	*Description*
Time step	Any positive real number	yes	n/a	The time step size for the simulation.
Number of time steps	Any non-negative integer	no	-1	The number of time steps until the simulation stops. Either this or the simulation end time must be specified. If both are specified, the simulation will end when the first condition is reached.
Simulation end time	Any non-negative real number	no	-0.1	The simulated time when until the simulation stops. Either this or the number of time steps must be specified. If both are specified, the simulation will end when the first condition is reached.

Elliptic Solver Parameters				
Name	*Options*	*Required*	*Default*	*Description*
Linear solver	SolverCG	no	SolverCG	Currently the only option is the deal.II conjugate gradient solver.
Use absolute convergence tolerance	Boolean	no	false	Sets whether to use an absolute tolerance for the linear solver (versus a relative tolerance).
Solver tolerance value	Any positive real number	no	1e-3	The tolerance for the linear solver. If an absolute tolerance is used, the iterative solver stops when the L2 norm of the residual drops below the tolerance. If a relative tolerance is used, the iterative solver stops when the L2 norm of the residual has dropped by a factor equal to the tolerance.
Maximum allowed solver iterations	Any positive integer	no	10000	The maximum number of iterations for the linear solver, if this number of iterations is reached, the solver stops regardless of the tolerance value.

Output				
Name	*Options*	*Required*	*Default*	*Description*
Output condition	EQUAL_SPACING, LOG_SPACING, N_PER_DECADE, LIST	no	EQUAL _SPACING	This sets the spacing between the times the simulation outputs the model variables. EQUAL_SPACING spaces them equally, LOG_SPACING spaces them $10^{n/(outputs)\log(timesteps)}$, N_PER_DECADE allows the user to set how many times the simulation outputs per power of ten iterations, and LIST outputs at a user-given list of time step numbers.
Domain size Y	Any positive real number	yes	n/a	The size of the domain in the y direction.
Domain size Z	Any positive real number	yes	n/a	The size of the domain in the z direction.

6 The Application Files: equations.h, ICs_and_BCs.h, postprocess.h, customPDE.h, and main.cc

6.1 parameters.h

All of the purely numerical parameters for an application are given in the file *parameters.h*. Each parameter is set using the C++ *#define* command. In C++ parlance, the statements are preprocessor macros. The compiler searches through the source code and replaces the first argument with the second argument. Here is a look at the parameters file for the coupled Allen-Cahn/Cahn-Hilliard example application:

```
// Parameter list for the coupled Allen−Cahn/Cahn−Hilliard example application
// All strictly numerical parameters should be set in this file

// ===================================================================
// Set the number of dimensions (1, 2, or 3 for a 1D, 2D, or 3D calculation)
// ===================================================================
#define problemDIM 2

// ===================================================================
// Set the length of the domain in all three dimensions
// ===================================================================
// Each axes spans from zero to the specified length
#define spanX 100.0
#define spanY 100.0
#define spanZ 100.0

// ===================================================================
// Set the element parameters
// ===================================================================
// The number of elements in each direction is 2^(refineFactor) * subdivisions
// For optimal performance, use refineFactor primarily to determine the element size
#define subdivisionsX 1
#define subdivisionsY 1
#define subdivisionsZ 1
#define refineFactor 7

// Set the polynomial degree of the element
// Suggested values are either 1 or 2
#define finiteElementDegree 1

// ===================================================================
// Set the time step parameters
// ===================================================================
// The size of the time step
#define timeStep 1.0e−3

// The simulation ends when either timeFinal is reached or the number of time steps
// equals timeIncrements
#define timeFinal 100.0
#define timeIncrements 100000

// ===================================================================
// Set the output parameters
// ===================================================================
// Each field in the problem will be output is writeOutput is set to "true"
#define writeOutput true

// Type of spacing between outputs ("EQUAL_SPACING", "LOG_SPACING", "N_PER_DECADE",
// or "LIST")
#define outputCondition "EQUAL_SPACING"

// Number of times the program outputs the fields (total number for "EQUAL_SPACING"
// and "LOG_SPACING", number per decade for "N_PER_DECADE", ignored for "LIST")
#define numOutputs 10

// User−defined list of time steps where the program should output. Only used if
// outputCondition is "LIST"
#define outputList {}

// Status is printed to the screen every skipPrintSteps
#define skipPrintSteps 10

// ===================================================================
// Set the flag determining if the total free energy is calculated for each output
// ===================================================================
#define calcEnergy true
```

In the first section of the parameters file, the user sets the number of dimensions for the simulation, an integer from 1 to 3. Strictly speaking, this is the only change the user needs to make for the calculation to go from a 1D simulation to a 2D simulation, to a 3D simulation. In practice, often the intial conditions and boundary conditions will have to change when the number of dimensions changes (see Section 6.3 for instructions on how to modify initial conditions or boundary conditions).

In the second section of the parameters file, the user sets the size of the computational domain. PRISMS-PF assumes that the domain is a line in 1D, a rectangle in 2D, and a rectangular prism in 3D. The length of the domain along the x, y, and z directions is given by *spanX*, *spanY*, and *spanZ*, respectively (as a floating point number). When *problemDIM* is less than 3, the unneeded values are ignored.

In the third section of the parameters file, the user sets parameters related to the finite elements. The first four parameters determine the number of elements in the simulation. The number of elements in each direction is given by $2^{refineFactor} \times subdivisions$. The three parameters

subdivisionsX, *subdivisionsY*, and *subdivisionsZ* determine how many times the domain is divided along each dimension (x, y, and z, respectively). The parameter *refineFactor* then determines how many times each of those divisions is bisected in each direction. For example, in 2D, if *subdivisionsX* is 1, *subdivisionsY* is 3, and *refineFactor* is 2, then there will be $2^2 \times 1 = 4$ elements in the x direction and $2^2 \times 3 = 12$ elements in the y direction. We find that using *refineFactor* to determine the element size results in improved perfomance compared to using subdivisions, particularly for parallel calculations. We recommend doing at least half of the refinement using *refineFactor*. For example, a combination of 5 subdivisions and a *refineFactor* of 3 should perform well, but a combination of 10 subdivsions and a *refineFactor* of 2 may not. The final parameter in this section is the polynomial degree of the basis functions for the elements (an integer). The order of accuracy (in space) is this number plus 1. For phase field applications, we recommend using values of 1 or 2. Generally speaking, increasing *finiteElementDegree* allows one to increase the element size without increasing the error. However, in a phase field calculation, the element size should be small enough that there is more than one element in the interface – effectively limiting *finiteElementDegree* to 2.

In the fourth section of the parameters file, the user sets parameters related to time stepping. Currently, PRISMS-PF is set up for forward Euler time stepping. The size of each time step is set by *timeStep* (a floating point number). The simulation finishes when either an amount of simulation time equal to *timeFinal* has passed, or *timeIncrements* time steps have occured.

In the fifth section of the parameters file, the user sets parameters related to the output. The first parameter in this section, *writeOutput*, is a flag that should be "true" if you want to output the results of the simulation and "false" if not. The output is written to the current directory. The second parameter, *outputCondition*, determines the type of spacing between outputs. The options are "EQUAL_SPACING", "LOG_SPACING", "N_PER_DECADE", and "LIST". The third parameter, *numOutputs*, determines the number of outputs. For any *outputCondition*, the initial conditions are outputted. For "EQUAL_SPACING", the progam will output *numOutputs* times after the initial conditions are output, separated by the total number of increments divded by *numOutputs*. For "LOG_SPACING", the progam will output *numOutputs* times after the initial conditions are output, where the increments where output occurs are given by:

$$10^{n/numOutputs \, \log(totalIncrements)} \tag{1}$$

rounded to the nearest integer, where n is the output number, and *totalIncrements* is the total number of time steps. For "N_PER_DECADE", the program will output *numOutput* times for each power of 10 after outputting the initial conditions. For example, if *numOutput* is 1, the program will output at every power of 10 (i.e. 1, 10, 100, 1000, ...). If *numOutput* is 10, the program will output ten times for every power of 10 (i.e. 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, ...). For "LIST", the user supplies a list of time steps where the program should output using the parameter *outputList*. *outputList* should be a bracketed, comma-separated list and is ignored for any type of output spacing other than "LIST". The final parameter in this section, *skipPrintSteps*, controls how often the status of the simulation prints to the screen. Setting *skipPrintSteps* to 1 will cause the simulation to print an update every time step; setting it to two will cause the simulation to print out every other time step, at 10 it will print every 10 time steps, etc.

In the sixth and final section of the parameters file, the user sets a flag determining if the integrated free energy should be calculated whenever the code outputs the results. If *calcEnergy* is "true", the total free energy will be written to a file names "freeEnergy.txt." If not given, the default value for *calcEnergy* is "false".

6.1.1 Parameters for Adaptive Meshing and Elliptic Equations

An examination of *parameters.h* in the Precipitate Evolution example application reveals two additional sections not in the parameters file for the coupled Allen-Cahn/Cahn-Hilliard example application:

```
// ====================================================================
// Set the adaptive mesh refinement parameters
// ====================================================================
// Set the flag determining if adaptive meshing is activated
#define hAdaptivity true

// Set the maximum and minimum level of refinement
#define maxRefinementLevel (refineFactor)
#define minRefinementLevel (refineFactor-2)

// Set the fields used to determine the refinement. Fields determined by the order
// declared in "equations.h", starting at zero
#define refineCriterionFields {1,2,3}

// Set the maximum and minimum value of the fields where the mesh should be refined
#define refineWindowMax {0.99,0.99,0.99}
#define refineWindowMin {0.01,0.01,0.01}

// Set the number of time steps between remeshing operations
#define skipRemeshingSteps 1000

[...]

// ====================================================================
// Set the elliptic solver parameters
// ====================================================================
// The solver type (currently the only recommended option is conjugate gradient)
#define solverType ''SolverCG''

// The flag that determines whether the tolerance for solver convergence should
// be an absolute tolerance (absTol=true) or a relative tolerance (absTol=false)
#define absTol true

// The tolerance for convergence (L2 norm of the residual)
#define solverTolerance 1.0e-4

// The maximum number of solver iterations per time step
#define maxSolverIterations 1000
```

The parameters in the first section govern adaptive meshing operations. If this section of parameters is omitted, a non-adaptive mesh is used. The first parameter in this section, *hAdaptivity*, is a flag determining if adaptive meshing will be activated. The second and third parameters, *maxRefinementLevel* and *minRefinementLevel*, determine the maximum and minimum levels of refinement alllowed. These values reflect local changes in the *refineFactor* parameter set earlier in the parameters file. The third parameter, *refineCriterionFields*, is a list of the fields used to determine how the mesh should be adapted. The fields should be in a comma-separated, bracketed list. Each field is referred to by its index, determined by the order it is declared in the "equations.h" file (discussed later), starting at zero. The fifth and sixth parameters, *refineWindowMax* and *refineWindowMin*, determine the upper and lower bounds of each field in the *refineCriterionFields* list that should be refined. In the above example, the mesh is refined wherever any of fields 1, 2, or 3 are between 0.01 and 0.99. Finally, the seventh parameter in this section, *skipRemeshingSteps*, sets the number of time steps between remeshing operations.

The parameters in the second section above govern the solution of elliptic PDEs, and are only needed for applications with at least one elliptic equation. In the Precipitate Evolution example application, these parameters are needed to solve for mechanical equilbrium. The first parameter is the type of solver used. Currently, conjugate gradient (*solverType*=SolverCG) is the only solver recommended for use. Other solvers in deal.ii could be used here (e.g. GMRES or MinRes), but will yield worse performance for the type of symmetric, positive-definite matrices encountered in most problems of interest. The second parameter is a flag that determines whether the tolerance for the convergence of the solver should be an absolute tolerance or a relative tolerance. The third parameter in this section is the tolerance for convergence. If *absTol* is "true", the iterative solver stops when the L_2 norm of the residual is less than *solverTolerance*. If *absTol* is"false", the iterative solver stops when the L_2 norm of the residual has decreased by a factor of *solverTolerance*

compared to the initial value of the residual. The fourth and final parameter in this section, *maxSolverIteration*, limits the number of solver iterations per time step. Once this number of iterations is reached, the solver stops, regardless of progress toward convergence.

6.2 equations.h

The file "equations.h" contains a list of the variables in the model equations and the residuals for the model equations. At the top of the file are a series of C++ *#define* commands. Later in the file are three functions: *residualRHS*, *residualLHS*, and *energyDensity*. Here is a look at the equations file for the coupled Allen-Cahn/Cahn-Hilliard example application:

```
// List of variables and residual equations for the coupled Allen-Cahn/Cahn-Hilliard example application

// =================================================================================
// Define the variables in the model
// =================================================================================
// The number of variables
#define num_var 2

// The names of the variables, whether they are scalars or vectors and whether the
// governing eqn for the variable is parabolic or elliptic
#define variable_name {"c", "n"}
#define variable_type {"SCALAR","SCALAR"}
#define variable_eq_type {"PARABOLIC","PARABOLIC"}

// Flags for whether the value, gradient, and Hessian are needed in the residual eqns
#define need_val {true, true}
#define need_grad {true, true}
#define need_hess {false, false}

// Flags for whether the residual equation has a term multiplied by the test function
// (need_val_residual) and/or the gradient of the test function (need_grad_residual)
#define need_val_residual {true, true}
#define need_grad_residual {true, true}

// =================================================================================
// Define the model parameters and the residual equations
// =================================================================================
// Parameters in the residual equations and expressions for the residual equations
// can be set here. For simple cases, the entire residual equation can be written
// here. For more complex cases with loops or conditional statements, residual
// equations (or parts of residual equations) can be written below in "residualRHS".

// Cahn-Hilliard mobility
#define McV 1.0

// Allen-Cahn mobility
#define MnV 150.0

// Allen-Cahn gradient energy coefficient
#define KnV 0.5

// Free energy for each phase and they're first and second derivatives
#define faV (-1.6704-4.776*c+5.1622*c*c-2.7375*c*c*c+1.3687*c*c*c*c)
#define facV (-4.776 + 10.3244*c - 8.2125*c*c + 5.4748*c*c*c)
#define faccV (10.3244-16.425*c+16.4244*c*c)
#define fbV (5.0*c*c-5.9746*c-1.5924)
#define fbcV (10.0*c-5.9746)
#define fbccV (10.0)

// Interpolation function and its derivative
#define hV (10.0*n*n*n-15.0*n*n*n*n+6.0*n*n*n*n*n)
#define hnV (30.0*n*n-60.0*n*n*n+30.0*n*n*n*n)

// Residual equations
#define muxV ( cx*((1.0-hV)*faccV+hV*fbccV) + nx*((fbcV-facV)*hnV) )
#define rcV    (c)
#define rcxV   (constV(-McV*timeStep)*muxV)
#define rnV   (n-constV(timeStep*MnV)*(fbV-faV)*hnV)
#define rnxV  (constV(-timeStep*KnV*MnV)*nx)

// =================================================================================
// residualRHS
// =================================================================================
// This function calculates the residual equations for each variable. It takes
// "modelVariablesList" as an input, which is a list of the value and derivatives of
// each of the variables at a specific quadrature point. The (x,y,z) location of
// that quadrature point is given by "q_point_loc". The function outputs
// "modelResidualsList", a list of the value and gradient terms of the residual for
// each residual equation. The index for each variable in these lists corresponds to
// the order it is defined at the top of this file (starting at 0).
template <int dim>
void generalizedProblem<dim>::residualRHS(const std::vector<modelVariable<dim>> & modelVariablesList,
        std::vector<modelResidual<dim>> & modelResidualsList,
        dealii::Point<dim, dealii::VectorizedArray<double> > q_point_loc) const {

// The concentration and its derivatives (names here should match those in the macros above)
scalarvalueType c = modelVariablesList[0].scalarValue;
```

26

```
scalargradType cx = modelVariablesList[0].scalarGrad;

// The order parameter and its derivatives (names here should match those in the macros above)
scalarvalueType n = modelVariablesList[1].scalarValue;
scalargradType nx = modelVariablesList[1].scalarGrad;

// Residuals for the equation to evolve the concentration (names here should match those in the macros above)
modelResidualsList[0].scalarValueResidual = rcV;
modelResidualsList[0].scalarGradResidual = rcxV;

// Residuals for the equation to evolve the order parameter (names here should match those in the macros above)
modelResidualsList[1].scalarValueResidual = rnV;
modelResidualsList[1].scalarGradResidual = rnxV;

}

// =================================================================================
// residualLHS (needed only if at least one equation is elliptic)
// =================================================================================
// This function calculates the residual equations for the iterative solver for
// elliptic equations.for each variable. It takes "modelVariablesList" as an input,
// which is a list of the value and derivatives of each of the variables at a
// specific quadrature point. The (x,y,z) location of that quadrature point is given
// by "q_point_loc". The function outputs "modelRes", the value and gradient terms of
// for the left-hand-side of the residual equation for the iterative solver. The
// index for each variable in these lists corresponds to the order it is defined at
// the top of this file (starting at 0), not counting variables that have
// "need_val_LHS", "need_grad_LHS", and "need_hess_LHS" all set to "false". If there
// are multiple elliptic equations, conditional statements should be used to ensure
// that the correct residual is being submitted. The index of the field being solved
// can be accessed by "this->currentFieldIndex".
template <int dim>
void generalizedProblem<dim>::residualLHS(const std::vector<modelVariable<dim>> & modelVariablesList,
        modelResidual<dim> & modelRes,
        dealii::Point<dim, dealii::VectorizedArray<double> > q_point_loc) const {

}

// =================================================================================
// energyDensity (needed only if calcEnergy == true)
// =================================================================================
// This function integrates the free energy density across the computational domain.
// It takes "modelVariablesList" as an input, which is a list of the value and
// derivatives of each of the variables at a specific quadrature point. It also
// takes the mapped quadrature weight, "JxW_value", as an input. The (x,y,z) location
// of the quadrature point is given by "q_point_loc". The weighted value of the
// energy density is added to "energy" variable and the components of the energy
// density are added to the "energy_components" variable (index 0: chemical energy,
// index 1: gradient energy, index 2: elastic energy).
template <int dim>
void generalizedProblem<dim>::energyDensity(const std::vector<modelVariable<dim>> & modelVariablesList,
        const dealii::VectorizedArray<double> & JxW_value,
        dealii::Point<dim, dealii::VectorizedArray<double> > q_point_loc) {

// The concentration and its derivatives (names here should match those in the macros above)
scalarvalueType c = modelVariablesList[0].scalarValue;
scalargradType cx = modelVariablesList[0].scalarGrad;

// The order parameter and its derivatives (names here should match those in the macros above)
scalarvalueType n = modelVariablesList[1].scalarValue;
scalargradType nx = modelVariablesList[1].scalarGrad;

// The homogenous free energy
scalarvalueType f_chem = (constV(1.0)-hV)*faV + hV*fbV;

// The gradient free energy
scalarvalueType f_grad = constV(0.5*KnV)*nx*nx;

// The total free energy
scalarvalueType total_energy_density;
total_energy_density = f_chem + f_grad;

// Loop to step through each element of the vectorized arrays. Working with deal.ii
// developers to see if there is a more elegant way to do this.
assembler_lock.acquire();
for (unsigned i=0; i<c.n_array_elements;i++){
  if (c[i] > 1.0e-10){
        this->energy+=total_energy_density[i]*JxW_value[i];
        this->energy_components[0]+= f_chem[i]*JxW_value[i];
        this->energy_components[1]+= f_grad[i]*JxW_value[i];
  }
}
assembler_lock.release();
}
```

This file is a bit longer than the parameters file. Unlike the parameters file, the entries in this file vary substantially from application to application.

In the first section of the equations file, the variables in the model are listed and properties regarding the variables and their governing equations are set. The first input, *num_var*, sets the number of variables in the model system. The second input in the section, *variable_name*, sets

the name of each of the variables. The names, as well as all of the other varable properties below, should be enclosed in quotes and in a comma-separated, bracketed list. The names are used to label the *vtu output file for the field. The third input in this section is for the type of each variable, either "SCALAR" or "VECTOR". The fourth input, *variable_eq_type*, if for the type of governing partial differential equation for each of the variables. Currently, the two options are "PARABOLIC" and "ELLIPTIC". The next three inputs in this section, *need_val*, *need_grad*, and *need_hess*, contain true/false flags for whether the value, gradient, or Hessian of that variable is needed in the residual equation. The final two inputs in this section, *need_val_residual* and *need_grad_residual*, contain true/false flags for whether the residual has terms proportional to the test function (*need_val_residual* = true) and/or the the residual has terms proportional to the gradient of the test function (*need_grad_residual* = true).

The second section of the equations file contains the definition of some global model parameters and the residual equations. Because the contents of this section vary substantially between applications, we won't explain it line-by-line, as in the previous sections. In the first few statements, several model parameters are set, including mobilities, the gradient energy coefficient, free energies and their derivatives, and the interpolation function and its derivative. Next, the residual expressions are given.

In PRISMS-PF, the residual equations are expressed in two terms. The first is the part of the integrand that is multiplied by the test function. The second is the part of the integrand that multipled by the gradient of the test function. For the coupled Cahn-Hilliard/Allen-Cahn system, the residual equations are

$$\int_\Omega w\eta^{n+1}\ dV = \int_\Omega w\left(\underbrace{\eta^n - \Delta tM_\eta\left((f_{\beta,c}^n - f_{\alpha,c}^n)H_{,\eta}^n\right)}_{r_\eta}\right) + \nabla w \cdot \underbrace{(-\Delta tM_\eta\kappa)\nabla\eta^n}_{r_{\eta x}}\ dV \qquad (2)$$

and

$$\int_\Omega wc^{n+1}\ dV = \int_\Omega w\underbrace{c^n}_{r_c} \qquad\qquad\qquad\qquad (3)$$

$$+ \nabla w\underbrace{(-\Delta tM_c)\ [\ (f_{\alpha,cc}^n(1 - H^{n+1}) + f_{\beta,cc}^n H^{n+1})\nabla c +\ ((f_{\beta,c}^n - f_{\alpha,c}^n)H_{,\eta}^{n+1}\nabla\eta)]}_{r_{cx}}\ dV$$

$$(4)$$

for the Allen-Cahn and Cahn-Hilliard equation, respectively. Each of the residual terms is marked with an underbrace. The r_η residual term is given by rnV in the equations file. The $r_{\eta x}$ residual term is given by $rnxV$ in the equations file. Similarly, the r_c and r_{cx} residual terms are given by rcV and $rcxV$, respectively.

The next section of the equations file contains a C++ function, named *residualRHS* where the residuals at a particular quadrature point are actually calculated. The inputs to this function are a list of the model variable values and derivatives, *modelVariablesList* and a point giving access to (x,y,z) coordinates, *q_point_loc*. The output is a list of the residual terms for each variable, *modelResidualsList*. The first few lines of the function set more convenient names for the variables and their derivatives. By convention, the value of the variable is denoted by the variable name (c for the concentration and n for the structural order parameter in this case), the list of first derivatives is denoted by the variable name followed by an "x", and second derivatives are denoted by the variable name followed by "xx". Each variable in *modelVariablesList* can be accessed by its

index – determined by the order it was declared at the top of the file, starting at zero. The variable value and the derivatives can be accessed through the *scalarValue*, *scalarGrad*, and *scalarHess* object members for scalar variables and the *vectorValue*, *vectorGrad*, and *vectorHess* object members for vector variables. The data type for the value of a scalar variable is *scalarvalueType* (a scalar), the data type for the first derivatives of a scalar variable is *scalargradType* (a vector with a length equal to the number of dimensions), and the data type for the second derivatives of a scalar variable is *scalarhessType* (a matrix with a size equal to the number of dimensions by the number of dimensions). For vector variables, the data types are *vectorvalueType* (a vector with length equal to the number of dimensions), *vectorgradType* (a matrix with a size equal to the number of dimensions by the number of dimensions), and *vectorgradType* (a rank-three tensor with a size in each direction equal to the number of dimensions).

A Note on Types: The deal.II library uses a data structure called a VectorizedArray to store the variable values and their derivatives. This data structured in optimized for modern vectorized processors, giving a substantial speedup in some cases. However, this data structure can complicate things slightly. One complicating factor is that VectorizedArrays can't always be added, subtracted, multiplied, or divided with more standard data types like doubles. For this reason, you will see the "constV(arguement)" function scattered throughout the code. This function turns a non-VectorizedArray into a VectorizedArray. To be safe, you can always encase non-VectorizedArrays with "constV()" when they share an operation with a VectorizedArray. A second complication is that not all of the standard mathematical operations are available for VectorizedArrays. The basic trigonometric functions are available, as are exponentials and square roots. However, hyperbolic tangents are not. If needed, they must be constructed from exponents. A third complication is that conditional statements involving VectorizedArrays are not allowed. VectorizedArrays contain data from multiple points that the processor operates on at once. Under normal conditions a conditional statement could only be applied to one of the points in the VectorizedArray. There are possible ways to break up the VectorizedArrays and work on the pieces of data individually, but this approach is not recommended for most users. Clever use of "max" and "min" functions (which do exist for VectorizedArrays) may be able to take the place of a conditional statement. For more details on the deal.II implementation of VectorizedArrays (including a list of mathematical operations that are allowed), please visit: https://www.dealii.org/8.4.0/doxygen/deal.II/classVectorizedArray.html

---

The names given to the variable values and derivatives in *residualRHS* should match what is used in the residual equations earlier in the file. Looking at the free energies, interpolation function, and the residuals, you see that *c*, *cx*, *n*, and *nx* are used repeatedly. Those usages refer to the variables defined in *residualRHS*.

After the names for the variable values and derivatives are set, you can add any terms that are too complicated to write as a macro up at the top of the file. Such terms would be terms with loops or conditional statements, or terms that warrant intermediate variables. The coupled Allen-Cahn/Cahn-Hilliard application is simple enough that it doesn't contain any such terms. Later in this section you will see that the Precipitate Evolution application does.

In the final part of this section, the residuals that were defined earlier in the file are used to set the residual terms in *modelResidualsList*. Like *modelVariablesList*, each residual is accessed by the index of the corresponding variable. The parts of the residual multiplied by the test function and the gradient of the test function are accessed through the *scalarValueResidual* and *scalarGradResidual* members for scalar variables and the *vectorValueResidual* and *vectorGradResidual* members for vector variables.

The next section is the function *residualLHS*. This function is only needed for systems of equations with at least one elliptic equation. The coupled Cahn-Hilliard/Allen-Cahn system doesn't have an elliptic equation, so it is left blank. You'll see *residualLHS* in action below in the Precipitate Evolution application.

The final section of the equations file is a function that calculates the total free energy of the system. This function is set up very similarly to *residualRHS*, starting with assigning convenient names to the variable names and derivatives in *modelVariablesList*. The difference is that instead of building up the residual equations, in this function one constructs the free energy density. In the code above, the homogenous free energy and gradient free energy are summed to get the total free energy density. At the bottom of the function is a loop to multiply the free energy density at a quadrature point by the mapped quadrature weight (given by *JxW_value*) and add that to the global energy. This portion of the code is a bit dirty – we're working with the deal.ii developers to find a better way to integrate the free energy density to get the free energy.

6.2.1 Systems with Elliptic Equations and Mechanical Properties

The equations file for the Precipitate Evolution example application is considerably more complex than the one for the coupled Cahn-Hilliard/Allen-Cahn example application. Three important differences is that the system of equations contains an elliptic equation, it needs mechanical properties defined, and it constructs some of the terms in the residual in *residualRHS*. Here are some excerpts of the equations file from the Precipitate Evolution example application:

```
// List of variables and residual equations for the Precipitate Evolution example application

// ================================================================
// Define the variables in the model
// ================================================================
// The number of variables
#define num_var 5

// The names of the variables, whether they are scalars or vectors and whether the
// governing eqn for the variable is parabolic or elliptic
#define variable_name {"c", "n1", "n2", "n3", "u"}
#define variable_type {"SCALAR","SCALAR","SCALAR","SCALAR","VECTOR"}
#define variable_eq_type {"PARABOLIC","PARABOLIC","PARABOLIC","PARABOLIC","ELLIPTIC"}

// Flags for whether the value, gradient, and Hessian are needed in the residual eqns
#define need_val {true, true, true, true, false}
#define need_grad {true, true, true, true, true}
#define need_hess {false, false, false, false, false}

// Flags for whether the residual equation has a term multiplied by the test function
// (need_val_residual) and/or the gradient of the test function (need_grad_residual)
#define need_val_residual {true, true, true, true, false}
#define need_grad_residual {true, true, true, true, true}

// Flags for whether the value, gradient, and Hessian are needed in the residual eqn
// for the left-hand-side of the iterative solver for elliptic equations
#define need_val_LHS {false, true, true, true, false}
#define need_grad_LHS {false, false, false, false, true}
#define need_hess_LHS {false, false, false, false, false}

// Flags for whether the residual equation for the left-hand-side of the iterative
// solver for elliptic equations has a term multiplied by the test function
// (need_val_residual) and/or the gradient of the test function (need_grad_residual)
#define need_val_residual_LHS {false, false, false, false, false}
#define need_grad_residual_LHS {false, false, false, false, true}

// ================================================================
// Define the model parameters and the residual equations
// ================================================================
// Parameters in the residual equations and expressions for the residual equations
// can be set here. For simple cases, the entire residual equation can be written
// here. For more complex cases with loops or conditional statements, residual
// equations (or parts of residual equations) can be written below in "residualRHS".

[...]

// Define mechanical properties
#define n_dependent_stiffness true
// Mechanical symmetry of the material and stiffness parameters
// If n_dependent_stiffness == false the first entry is used for all phases
#define MaterialModels {{"ISOTROPIC"},{"ISOTROPIC"}}
#define MaterialConstants {{2.0,0.3},{2.5,0.3}}

[...]
```

```
// Residual equations
#define rcV    (c)
#define rcxTemp ( cx*((1.0-h1V-h2V-h3V)*faccV+(h1V+h2V+h3V)*fbccV)
                    + n1x*((fbcV-facV)*hn1V) + n2x*((fbcV-facV)*hn2V) + n3x*((fbcV-facV)*hn3V) + grad_mu_el)
#define rcxV   (constV(-timeStep*McV)*rcxTemp)


#define rn1V    (n1-constV(timeStep*Mn1V)*((fbV-faV)*hn1V+nDependentMisfitAC1+heterMechAC1))
#define rn2V    (n2-constV(timeStep*Mn2V)*((fbV-faV)*hn2V+nDependentMisfitAC2+heterMechAC2))
#define rn3V    (n3-constV(timeStep*Mn3V)*((fbV-faV)*hn3V+nDependentMisfitAC3+heterMechAC3))
#define rn1xV   (constV(-timeStep*Mn1V)*Knx1)
#define rn2xV   (constV(-timeStep*Mn2V)*Knx2)
#define rn3xV   (constV(-timeStep*Mn3V)*Knx3)

// =================================================================================
// residualRHS
// =================================================================================
// This function calculates the residual equations for each variable. It takes
// "modelVariablesList" as an input, which is a list of the value and derivatives of
// each of the variables at a specific quadrature point. The (x,y,z) location of
// that quadrature point is given by "q_point_loc". The function outputs
// "modelResidualsList", a list of the value and gradient terms of the residual for
// each residual equation. The index for each variable in these lists corresponds to
// the order it is defined at the top of this file (starting at 0).
template <int dim>
void generalizedProblem<dim>::residualRHS(const std::vector<modelVariable<dim>> & modelVariablesList,
                        std::vector<modelResidual<dim>> & modelResidualsList,
                        dealii::Point<dim, dealii::VectorizedArray<double> > q_point_loc) const {

// The concentration and its derivatives (names here should match those in the macros above)
scalarvalueType c = modelVariablesList[0].scalarValue;
scalargradType cx = modelVariablesList[0].scalarGrad;

// The first order parameter and its derivatives (names here should match those in the macros above)
scalarvalueType n1 = modelVariablesList[1].scalarValue;
scalargradType n1x = modelVariablesList[1].scalarGrad;

// The second order parameter and its derivatives (names here should match those in the macros above)
scalarvalueType n2 = modelVariablesList[2].scalarValue;
scalargradType n2x = modelVariablesList[2].scalarGrad;


// The third order parameter and its derivatives (names here should match those in the macros above)
scalarvalueType n3 = modelVariablesList[3].scalarValue;
scalargradType n3x = modelVariablesList[3].scalarGrad;

// The derivative of the displacement vector (names here should match those in the macros above)
vectorgradType ux = modelVariablesList[4].vectorGrad;
vectorgradType ruxV;

[...]

if (n_dependent_stiffness == true){
dealii::VectorizedArray<double> sum_hV;
sum_hV = h1V+h2V+h3V;
for (unsigned int i=0; i<2*dim-1+dim/3; i++){
        for (unsigned int j=0; j<2*dim-1+dim/3; j++){
                CIJ_combined[i][j] = this->userInputs.CIJ_list[0][i][j]*(constV(1.0)-sum_hV)
                                        + this->userInputs.CIJ_list[1][i][j]*sum_hV;
        }
}
computeStress<dim>(CIJ_combined, E2, S);
}
else{
computeStress<dim>(this->userInputs.CIJ_list[0], E2, S);
}

// Fill residual corresponding to mechanics
// R=-C*(E-E0)

for (unsigned int i=0; i<dim; i++){
for (unsigned int j=0; j<dim; j++){
        ruxV[i][j] = - S[i][j];
}
}

[...]

modelResidualsList[0].scalarValueResidual = rcV;
modelResidualsList[0].scalarGradResidual = rcxV;

modelResidualsList[1].scalarValueResidual = rn1V;
modelResidualsList[1].scalarGradResidual = rn1xV;

modelResidualsList[2].scalarValueResidual = rn2V;
modelResidualsList[2].scalarGradResidual = rn2xV;

modelResidualsList[3].scalarValueResidual = rn3V;
modelResidualsList[3].scalarGradResidual = rn3xV;

modelResidualsList[4].vectorGradResidual = ruxV;

}

// =================================================================================
// residualLHS (needed only if at least one equation is elliptic)
// =================================================================================
// This function calculates the residual equations for the iterative solver for
```

```
// elliptic equations.for each variable. It takes "modelVariablesList" as an input,
// which is a list of the value and derivatives of each of the variables at a
// specific quadrature point. The (x,y,z) location of that quadrature point is given
// by "q_point_loc". The function outputs "modelRes", the value and gradient terms of
// for the left-hand-side of the residual equation for the iterative solver. The
// index for each variable in these lists corresponds to the order it is defined at
// the top of this file (starting at 0), not counting variables that have
// "need_val_LHS", "need_grad_LHS", and "need_hess_LHS" all set to "false". If there
// are multiple elliptic equations, conditional statements should be used to ensure
// that the correct residual is being submitted. The index of the field being solved
// can be accessed by "this->currentFieldIndex".
template <int dim>
void generalizedProblem<dim>::residualLHS(const std::vector<modelVariable<dim>> & modelVariablesList,
                modelResidual<dim> & modelRes,
                dealii::Point<dim, dealii::VectorizedArray<double> > q_point_loc) const {

//n1
scalarvalueType n1 = modelVariablesList[0].scalarValue;

//n2
scalarvalueType n2 = modelVariablesList[1].scalarValue;

//n3
scalarvalueType n3 = modelVariablesList[2].scalarValue;

//u
vectorgradType ux = modelVariablesList[3].vectorGrad;
vectorgradType ruxV;

// Take advantage of E being simply 0.5*(ux + transpose(ux)) and use the dealii "symmetrize" function
dealii::Tensor<2, dim, dealii::VectorizedArray<double> > E;
E = symmetrize(ux);

// Compute stress tensor (which is equal to the residual, ruxV)
if (n_dependent_stiffness == true){
        dealii::Tensor<2, CIJ_tensor_size, dealii::VectorizedArray<double> > CIJ_combined;
        CIJ_combined = this->userInputs.CIJ_list[0]*(constV(1.0)-h1V-h2V-h3V) + this->userInputs.CIJ_list[1]*(h1V+h2V+h3V);

        computeStress<dim>(CIJ_combined, E, ruxV);
}
else{
        computeStress<dim>(this->userInputs.CIJ_list[0], E, ruxV);
}

modelRes.vectorGradResidual = ruxV;

}

[...]
```

Because the system of equations in Precipitate Evolution contains an elliptic equation, new sections of the code are needed. In the declaration of the variables in the system of equations at the top of the file, five new variables are needed: *need_val_LHS*, *need_grad_LHS*, *need_hess_LHS*, *need_val_residual_LHS*, and *need_grad_residual_LHS*. Like the flags for *residualRHS*, these flags are set based on the need for the value, first derivatives, second derivatives of the variables and whether each of the two types of residual terms are needed, this time for the residual equations for the left-hand-side of the iterative solver for the elliptic equation(s). Here you also see an example of the declaration of a vector model variable (the displacement field in this case).

In the second section of the file, two variables related to the mechanical properties of the materials are set: *MaterialModels* and *MaterialConstants*. The variable *MaterialModels* contains a list of the symmetry of the stiffness tensor for all of the materials in the system. The options are: "ISOTROPIC", "TRANSVERSE", "ORTHOTROPIC", and "ANISOTROPIC". The symmetry for each material should be an upper-case string encased in quotes and brackets. The list is then put into a comma-separated, bracketed list. The variable *MaterialConstants* must give enough information to build the full stiffness matrix for the material. The number of constants and their meaning is different for different symmetry types:

- ISOTROPIC (2D/3D): 2 constants {Young's Modulus, Poisson's Ratio}
- TRANSVERSE (3D): 5 constants {C_{11}, C_{33}, C_{44}, C_{12}, C_{13}}
- ORTHOTROPIC (3D): 9 constants {C_{11}, C_{22}, C_{33}, C_{44}, C_{55}, C_{66}, C_{12}, C_{13}, C_{23}}
- ANISOTROPIC (3D): 21 constants {C_{11}, C_{22}, C_{33}, C_{44}, C_{55}, C_{66}, C_{12}, C_{13}, C_{14}, C_{15}, C_{16},

C_{23}, C_{24}, C_{25}, C_{26}, C_{34}, C_{35}, C_{36}, C_{45}, C_{46}, C_{56}}

- ANISOTROPIC (2D): 6 constants {C_{11}, C_{22}, C_{66}, C_{12}, C_{16}, C_{26}}

where C_{ij} refers to elements of the stiffness matrix.

Still in the second section of the file, the residual terms for the Cahn-Hilliard equation and three Allen-Cahn equations are set. The residual term for the displacement field isn't set here. It won't be set until the next section, inside *residualRHS*.

The third section of the file is the function *residualRHS*. In the code excerpt above, there are several references to the variable *this-¿userInputs.CIJ_list*. This is a vector of the stiffness matrices generated from the *MaterialModels* and *MaterialConstants* inputs. The first index for *CIJ_list* denotes the position in the list (i.e. 0 for the first material properties given in *MaterialModels* and *MaterialConstants*, 1 for the second material properties given, etc.). The individual elements of the stiffness matrix are accessed using the second and third indices. Therefore, if you wanted to access the C_{12} element for the first material in the list, you could access it through *CIJ_list*[0][1][2]. This example application also shows that the residuals can be defined in *residualsRHS* instead of as a macro in the second section of the file. The variable *ruxV* is set to minus the stress on an element-by-element basis. The *ruxV* residual is treated the same as the other residual terms at the end of the function, where they are loaded into *modelResidualsList*.

Unlike the coupled Cahn-Hilliard/Allen-Cahn example application, the Precipitate Evolution example application has calculations inside *residualLHS*. This function contains the residual on the left-hand-side of equation for the iterative solution of an elliptic equation using a conjugate gradient solver. The solution u is found through the following expression, given an initial guess u_0:

$$0 = R(u) = R(u_0 + \Delta u) \tag{5}$$

where $\Delta u = u - u_0$. Then, applying the discretization that $u = \sum_i w^i U^i$, we can write the following linearization:

$$\frac{\delta R(u)}{\delta u} \Delta U = -R(u_0) \tag{6}$$

The discretized form of this equation can be written as a matrix inversion problem. However, in PRISMS-PF, we only care about the product $\frac{\delta R(u)}{\delta u} \Delta U$. Taking the variational derivative of $R(u)$ yields:

$$\frac{\delta R(u)}{\delta u} = \frac{d}{d\alpha} \int_\Omega \nabla w : C : \left[\epsilon(u + \alpha w) - \epsilon^0 \right] \ dV \bigg|_{\alpha=0} \tag{7}$$

$$= \int_\Omega \nabla w : C : \frac{1}{2} \frac{d}{d\alpha} \left[\nabla(u + \alpha w) + \nabla(u + \alpha w)^T - \epsilon^0 \right] \ dV \bigg|_{\alpha=0} \tag{8}$$

$$= \int_\Omega \nabla w : C : \frac{d}{d\alpha} \left[\nabla(u + \alpha w) - \epsilon^0 \right] \ dV \bigg|_{\alpha=0} \quad (\textit{due to the symmetry of } C) \tag{9}$$

$$= \int_\Omega \nabla w : C : \nabla w \ dV \tag{10}$$

In its discretized form $\frac{\delta R(u)}{\delta u} \Delta U$ is:

$$\frac{\delta R(u)}{\delta u} \Delta U = \sum_i \sum_j \int_\Omega \nabla N^i : C : \nabla N^j dV \ \Delta U^j \tag{11}$$

33

Moving back to the non-discretized form yields:

$$\frac{\delta R(u)}{\delta u}\Delta U = \int_\Omega \nabla w : C : \nabla(\Delta u)dV \tag{12}$$

Thus, the full equation relating u_0 and Δu is:

$$\int_\Omega \nabla w : \underbrace{C : \nabla(\Delta u)}_{r_{ux}^{LHS}} dV = -\int_\Omega \nabla w : \underbrace{\sigma}_{r_{ux}} dV \tag{13}$$

The above values of r_{ux}^{LHS} and r_{ux} are used to define the residuals in the equations.h file. A similar process can be undertaken for other elliptic problems.

Like *residualRHS*, *residualLHS* takes *modelVariablesList* and *q_point_loc* as inputs. However, because only one elliptic equation is solved at a time, the output is a single residual variable, *modelRes*. As in *residualRHS*, the model variable values and derivatives are given convenient names at the start of the file. In *residualLHS*, *modelVariablesList* only contains variables where at least one of *need_val_LHS*, *need_grad_LHS*, or *need_hess_LHS* is "true". In this example, only the three structural order parameters (n1, n2, and n3) and the displacement field (u) are needed to calculate the LHS residual. Therefore, the indices are shifted from those in *residualRHS*, since the concentration at index 0 isn't needed. Similar to *residualRHS*, the value of *ruxV* is set and then used to update the appropriate member of *modelRes* (in this case *vectorGradResidual*).

If multiple elliptic equations need to be solved, you will need to use conditional statements to calculate the proper residual depending on the elliptic equation being solved. The index of the field being solved can be accessed using the "this->currentFieldIndex" statement. This index refers to the variable's overall order in the declaration, **not** the index used for *modelVariablesList* in *residualLHS*.

6.3   ICs_and_BCs.h

The final file that most users will interact with is the initial conditions and boundary conditions file, "ICs_and_BCs.h". The first two functions in the file, *InitialCondition* and *InitialCondition-Vec* are used to set the initial conditions for scalar and vector variables, respectively. The third function, *setBCs*, sets the boundary conditions for all of the model variables. Here's a look at "ICs_and_BCs.h" for the coupled Allen-Cahn/Cahn-Hilliard example application:

```
template <int dim>
class InitialCondition : public Function<dim>
{
public:
  unsigned int index;
  Vector<double> values;
  InitialCondition (const unsigned int _index) : Function<dim>(1), index(_index) {
    std::srand(Utilities::MPI::this_mpi_process(MPI_COMM_WORLD)+1);
  }
  double value (const Point<dim> &p, const unsigned int component = 0) const
  {
        double scalar_IC = 0;
        // =====================================================
        // ENTER THE INITIAL CONDITIONS HERE FOR SCALAR FIELDS
        // =====================================================
        // Enter the function describing conditions for the fields at point "p".
        // Use "if" statements to set the initial condition for each variable
        // according to its variable index.

        double dx=spanX/((double) subdivisionsX)/std::pow(2.0,refineFactor);
        double r=0.0;

        // Initial condition for the concentration field
        if (index == 0){
                if (dim == 2){
                        r=p.distance(Point<dim>(spanX/3.0,spanY/3.0));
                        scalar_IC = 0.009+0.5*(0.125)*(1.0-std::tanh((r-spanX/5.0)/(3*dx)));
                        r=p.distance(Point<dim>(3.0*spanX/4.0,3.0*spanY/4.0));
```

```cpp
                        scalar_IC += 0.5*(0.125)*(1.0 − std::tanh((r−spanX/12.0)/(3*dx)));
                    }
                    else if (dim == 3) {
                        r=p.distance(Point<dim>(spanX/3.0,spanY/3.0,spanZ/3.0));
                        scalar_IC = 0.009+0.5*(0.125)*(1.0 − std::tanh((r−spanX/5.0)/(3*dx)));
                        r=p.distance(Point<dim>(3.0*spanX/4.0,3.0*spanY/4.0,3.0*spanZ/4.0));
                        scalar_IC += 0.5*(0.125)*(1.0 − std::tanh((r−spanX/12.0)/(3*dx)));
                    }

                }
                // Initial condition for the structural order parameter field
                else {
                    if (dim == 2){
                        r=p.distance(Point<dim>(spanX/3.0,spanY/3.0));
                        scalar_IC = 0.5*(1.0 − std::tanh((r−spanX/5.0)/(3*dx)));
                        r=p.distance(Point<dim>(3.0*spanX/4.0,3.0*spanY/4.0));
                        scalar_IC += 0.5*(1.0 − std::tanh((r−spanX/12.0)/(3*dx)));
                    }
                    else if (dim == 3){
                        r=p.distance(Point<dim>(spanX/3.0,spanY/3.0,spanZ/3.0));
                        scalar_IC = 0.5*(1.0 − std::tanh((r−spanX/5.0)/(3*dx)));
                        r=p.distance(Point<dim>(3.0*spanX/4.0,3.0*spanY/4.0,3.0*spanZ/4.0));
                        scalar_IC += 0.5*(1.0 − std::tanh((r−spanX/12.0)/(3*dx)));
                    }
                }

            // =================================================================
            return scalar_IC;
        }
};

//initial condition
template <int dim>
class InitialConditionVec : public Function<dim>
{
public:
    unsigned int index;
    //Vector<double> values;
    InitialConditionVec (const unsigned int _index) : Function<dim>(dim), index(_index) {
        std::srand(Utilities::MPI::this_mpi_process(MPI_COMM_WORLD)+1);
    }
    void vector_value (const Point<dim> &p, Vector<double> &vector_IC) const
    {
        // =================================================================
        // ENTER THE INITIAL CONDITIONS HERE FOR VECTOR FIELDS
        // =================================================================
        // Enter the function describing conditions for the fields at point "p".
        // Use "if" statements to set the initial condition for each variable
        // according to its variable index.


        // =================================================================
    }
};

template <int dim>
void generalizedProblem<dim>::setBCs(){

    // =================================================================
    // ENTER THE BOUNDARY CONDITIONS HERE
    // =================================================================
    // This function sets the BCs for the problem variables
    // The function "inputBCs" should be called for each component of
    // each variable and should be in numerical order. Four input arguments
    // set the same BC on the entire boundary. Two plus two times the
    // number of dimensions inputs sets separate BCs on each face of the domain.
    // Inputs to "inputBCs":
    // First input: variable number
    // Second input: component number
    // Third input: BC type (options are "ZERO_DERIVATIVE", "DIRICHLET", and "PERIODIC")
    // Fourth input: BC value (ignored unless the BC type is "DIRICHLET")
    // Odd inputs after the third: BC type
    // Even inputs after the third: BC value
    // Face numbering: starts at zero with the minimum of the first direction, one for the
    //                 maximum of the first direction two for the minimum of the second
    //                 direction, etc.

    inputBCs(0,0,"ZERO_DERIVATIVE",0);
    inputBCs(1,0,"ZERO_DERIVATIVE",0);

}
```

Most users should only need to modify the sections between the lines. The first section sets initial conditions for scalar fields. The variable *scalarIC* should be set to the desired intitial condition. The (x,y,z) coordinates can be accessed by the p variable, with indices zero through 2. For example, if you wanted to initialize a field to $5xy$, you would enter:

```cpp
scalarIC = 5.0 * p[0] * p[1];
```

The initial conditions given above are a bit more complicated. They set the initial conditions for two circular (spherical in 3D) particles. The initial condition for different variables is set using

conditional statements and the *index* variable. The *index* variable refers to the order that the variable is declared in "equations.h". In this file, the deal.II function *distance* is used to find the distance between the point p and a specified point (the center of the circle/sphere). This example uses conditional statements to set the initial conditions differently for the concentration (*index*==0) and the structural order parameter (*index*==0) .

The second section is empty for this application because it doesn't have any vector variables.

The third and final section of the file is used to set the boundary conditions. This file should contain a list of *inputBCs* statements, one for each component of each variable. In this example application, there are two scalar variables, so two *inputBCs* statements are needed. The *inputBCs* statement can be used to set all boundaries to the same boundary condition or to set the boundary condition for each boundary seperately. The *inputBCs* statements for each variable should be given in the order that the variables were declared in "equations.h". For vector variables, the *inputBCs* statement for each component should be given consecutively. Currently, PRISMS-PF assumes that users will want to set a uniform boundary condition along each edge/face of the domain. More general cases can be treated using deal.II and could be added to PRISMS-PF if needed.

To set the same boundary condition on all boundaries for a variable, the *inputBCs* statement should have four arguements. The first argument is the variable index (once again determined by the order it is declared in the "equations.h" file). The second is the component of that variable (starting at zero). The third arguement is the type of boundary condition. Currently, the three options are and "ZERO_DERIVATIVE", "PERIODIC", and "DIRICHLET". Nonzero Neumann boundary conditions need to be applied as part of the residual equations, not through *setBCs*. The fouth arguement is the boundary condition value, which is only used for Dirichlet boundary conditions.

For different boundary conditions on each face/edge of the domain, the number of arguements for the *inputBCs* statement should have two plus two times the number of dimensions (4 in 1D, 6 in 2D, 8 in 3D). The first two arguments are still the variable index and component index. The next arguments come in pairs of boundary condition type and boundary condition value, repeated as many times as needed for the number of dimensions. The boundary condition from the first of these type/value pairs will be applied to the face/edge on the minimum of the x axis. The boundary condition for the second will be applied to the face/edge on the maximimum of the x axis. In 2D/3D, the third and fourth pairs will determine the boundary conditions on the minimum and maximum of the y axis. Similarly, in 3D, the fifth and sixth pairs will determine the boundary conditions on the minimum and maximum of the z axis. Thus in 2D, if one wanted a Dirichlet boundary condition of 1.0 on the x axis minimum, a Dirichlet boundary condition of -1.0 on the x axis maximum and zero derivative boundary conditions on the other two edges, one would enter:

```
inputBCs(0,0,"DIRICHLET",1.0"DIRICHLET",−1.0,"ZERO_DERIVATIVE",0,"ZERO_DERIVATIVE",0);
```

6.3.1   Initial Conditions and Boundary Conditions for Vector Variables

The main difference between "ICs_and_BCs.h" for the coupled Allen-Cahn/Cahn-Hilliard application and the Precipitate Evolution application is the existence of vector variables in the Precipitate Evolution application. Here are the last two functions from "ICs_and_BCs.h" in the Precipitate Evolution application:

```
[...]
//initial condition
template <int dim>
class InitialConditionVec : public Function<dim>
{
public:
```

36

```cpp
unsigned int index;
//Vector<double> values;
InitialConditionVec (const unsigned int _index) : Function<dim>(dim), index(_index) {
    std::srand(Utilities::MPI::this_mpi_process(MPI_COMM_WORLD)+1);
}
void vector_value (const Point<dim> &p, Vector<double> &vector_IC) const
{
        // ================================================================
        // ENTER THE INITIAL CONDITIONS HERE FOR VECTOR FIELDS
        // ================================================================
        // Enter the function describing conditions for the fields at point "p".
        // Use "if" statements to set the initial condition for each variable
        // according to its variable index.

        if (index==4){
                vector_IC(0) = 0.0;
                vector_IC(1) = 0.0;
                if (dim == 3){
                        vector_IC(2) = 0.0;
                }
        }
        // ================================================================
}
};

template <int dim>
void generalizedProblem<dim>::setBCs(){

        // ================================================================
        // ENTER THE BOUNDARY CONDITIONS HERE
        // ================================================================
        // This function sets the BCs for the problem variables
        // The function "inputBCs" should be called for each component of
        // each variable and should be in numerical order. Four input arguments
        // set the same BC on the entire boundary. Two plus two times the
        // number of dimensions inputs sets separate BCs on each face of the domain.
        // Inputs to "inputBCs":
        // First input: variable number
        // Second input: component number
        // Third input: BC type (options are "ZERO_DERIVATIVE", "DIRICHLET", and "PERIODIC")
        // Fourth input: BC value (ignored unless the BC type is "DIRICHLET")
        // Odd inputs after the third: BC type
        // Even inputs after the third: BC value
        // Face numbering: starts at zero with the minimum of the first direction, one for the maximum of the first direction,
        //                 two for the minimum of the second direction, etc.
        inputBCs(0,0,"ZERO_DERIVATIVE",0);

        inputBCs(1,0,"ZERO_DERIVATIVE",0);

        inputBCs(2,0,"ZERO_DERIVATIVE",0);

        inputBCs(3,0,"ZERO_DERIVATIVE",0);

        inputBCs(4,0,"DIRICHLET",0.0);
        inputBCs(4,1,"DIRICHLET",0.0);
        if (dim == 3){
                inputBCs(4,2,"DIRICHLET",0.0);
        }

}
```

Vector initial conditions are set much like scalar initial conditions. The initial conditions are set using the variable *vector_IC*. Each component of *vector_IC* is accessed using parentheses. As with scalar initial conditions, conditional statements with the variable index are used to set the initial condition for the proper variable.

Vector boundary conditions are set much like scalar initial conditions. Separate boundary conditions are specified for each component of the variable. The boundary conditions for the components should be listed in order (x then y then z). The boundary condition for each component of a vector variable is set exactly like that of a scalar variable.

7   Advanced Files in the Application Directory

7.1   main.cc

The file main.cc is generally boilerplate code. Its purpose is to drive the primary flow of the code. The following code snippet is the core of the file:

```cpp
userInputParameters<problemDIM> userInputs;
```

```
userInputs.loadUserInput();

customPDE<problemDIM,finiteElementDegree> problem(userInputs);

 problem.setBCs();
 problem.buildFields();
 problem.init  ();
 problem.solve();
```

In this portion of the code there are two main objects (in the C++ sense) of interest: userInput-Parameters and customPDE. The userInputParameters object reads and stores the inputs from the input files. These inputs are used in the creation of a customPDE object, which then is used to solve the desired set of equations. After the customPDE object is created, the setBCs, build-Fields, init, and solve methods (i.e. functions) are called to prepare the compuational system, solve the equations, and output the solution.

At the top of main.cc is a list of header files to be included and at the bottom is some code to handle errors during execution. It is rare that a user would need to modify these sections. For nearly all of the example applications, the main.cc file is identical. Two sets of exceptions are discussed below.

The applications that include nucleation (_nucleationModel and _preferential_nucleationModel) have three more lines in the list of included headers:

```
#include <random>
#include <time.h>
#include "nucleation.h"
```

The allenCahn_pfield and precipitateEvolution_pfunction applications have some extra lines in the list of included headers to utilize that functionality.

7.2   customPDE.h

As seen in main.cc, an application creates a customPDE object. Each application has its own customPDE class that is specified in customPDE.h. For most of the example applications, customPDE.h is the same, just containing a constructor, the setBCs function used in ICs_and_BCs.h, and the three functions in equations.h. However, the user is free to define any new variables or methods needed for their application as part of this class. Most importantly, since customPDE is a subclass of MatrixFreePDE, the primary solver class in the core library, methods in customPDE can be written to override the default behavior of the core library (all methods in MatrixFreePDE except for the constructor are virtual). A user may want to do this for functionality that isn't currently present in the core library (i.e. a new condition for mesh adaptivity), for performance (stripping out unneccesary evaluations for that given application), or to use PRISMS-PF in ways that we didn't plan for. In the applications with nucleation, the list of nuclei and the methods for nucleation are members of customPDE.

8   Postprocessing

PRISMS-PF includes a postprocessing suite to help analyze the simulation results. Often, it is easier to perform the postprocessing using PRISMS-PF, rather than a program such as VisIt or ParaView, especially when dealing with spatial derivatives. Postprocessing functionality is set in the postprocess.h file. The structure of this file is much like equations.h. For the remainder of this section, we will walk through the postprocess.h file from the allenCahn application.

The first section is much like the first section of equations.h, where the problem variables are defined. In this case, the postprocessing variables are defined (i.e. the variables other than those listed in equations.h that you would like to output). The first two blocks contain *pp_num_var*, the number of postprocessed variables, *pp_variable_name*, the name assigned to the variable in the *.vtu file, and *pp_variable_type*, whether the variable is a scalar or a vector. Currently only scalar postprocessing variables are allowed. In the next block, the user specifies if the value, gradient or hessian <u>of the model variables</u> is needed to calculate the postprocessed variables. In the last block, the user selects whether each postprocessed variable needs a value residual and/or a gradient residual.

```
// ===========================================================
// Define the variables for postprocessing
// ===========================================================
// The number of variables
#define pp_num_var 1

// The names of the variables, whether they are scalars or vectors
#define pp_variable_name {"mag_grad_n"}
#define pp_variable_type {"SCALAR"}

// Flags for whether the value, gradient, and Hessian are needed in the residual eqns
#define pp_need_val {true}
#define pp_need_grad {true}
#define pp_need_hess {false}

// Flags for whether the residual equation has a term multiplied by the test function
// (need_val_residual) and/or the gradient of the test function (need_grad_residual)
#define pp_need_val_residual {true}
#define pp_need_grad_residual {false}

// ===========================================================
```

The remainder of the file is the function postProcessedFields:

```
template <int dim>
void postProcessedFields(const std::vector<modelVariable<dim> > & modelVariablesList,
                         std::vector<modelResidual<dim> > & modelResidualsList,
                         const dealii::Point<dim, dealii::VectorizedArray<double> > q_point_loc,
                         const list_of_CIJ<dim> material_moduli) {

// The order parameter and its derivatives (names here should match those in the macros above)
dealii::VectorizedArray<double> n = modelVariablesList[0].scalarValue;
dealii::Tensor<1, dim, dealii::VectorizedArray<double> > nx = modelVariablesList[0].scalarGrad;

dealii::Tensor<1, dim, dealii::VectorizedArray<double> > pp_field;
pp_field[0] = nx[0];
pp_field[1] = nx[1];


// Residuals for the equation to evolve the order parameter (names here should match those in the macros above)
modelResidualsList[0].scalarValueResidual = std::sqrt(pp_field[0]*pp_field[0]+pp_field[1]*pp_field[1]); //constV(0.0);

}
```

The structure of this function is identical to that of residualRHS in equations.h.

9 Creating Custom Applications

You can create your own PRISMS-PF applications by making simple modifications to the input files described in the previous section. To create a new application, simply create a new folder in the applications directory and copy the files from a related example application. **If the file "CMakeCache.txt" exists, delete it** (this file contains paths to the original application, and if left would cause the compiler to compile the code from the original application). Then, make your desired modifications (changing the initial or boundary conditions, changing the residual equations, etc.) and compile and run your application the same way you would one of the example applications.

Once your have your own applications running, we'd love to help you share them with the community. Please contact us at prismsphasefield.dev@umich.edu if you need help using GitHub to generate a pull request so that we can add your work to the public repository.